6 Test-Driven Development（テスト駆動開発）開発手法としてのテスト

大月美佳
佐賀大学

テスト駆動開発とは

テスト駆動開発（Test-Driven Development（TDD））とは，単体テストを設計のために使う開発技法である．今回の特集のほかの記事では，主としてテスターのためのテスト技法を扱っているが，本稿では主としてプログラマのための開発技法としてテスト駆動開発を紹介する．

テスト駆動開発は K. Beck が 2003 年に「TDD：Test-Driven Development By Example」という書籍を出して世に広まった．この書籍は同年に日本でも「テスト駆動開発入門」として発売された[1]．日本ではアジャイル開発プロセスに対する関心が高かったため，比較的早期にテスト駆動開発も紹介された．現在も関連文献は比較的早期に翻訳される傾向にある．

K. Beck はまた E. Gamma と共に，テスト駆動開発を支援するツールとして，JUnit[2] を開発し提供を行った．この支援ツールが使いやすかったこともあって，テスト駆動開発は広く受け入れられ，今もなお注目を集めている．たとえば，文献 3）では，テスト駆動開発を特集している．

テスト駆動開発ではプログラマが単体テストを利用しながら，徐々にプログラムのコードを変更・適応させていく．ここで言う適応とは，コードをその時点で必要とされる程度に機能拡張していくことを言う．「リファクタリング」[4] の著者である M. Fowler は，テスト駆動開発を「テスト・ファースト（Test First）＋リファクタリング（Refactoring）」であると述べている．ここで，テスト・ファーストはそのまま「テストを最初に書く」ということで，リファクタリングとは「外部的な振る舞いを変えずに内部実装を書き換えていく」ことを言う．単にテストを最初に書くだけではなく，コードをリファクタリングして徐々に機能拡張していくのがテスト駆動開発であると言える．

この徐々にプログラムを機能拡張していくという性質から，テスト駆動開発は元々アジャイルな開発プロセスの 1 つである XP（eXtreme Programming）の実践項目の 1 つとして提案されていた．つまり，XP は仕様変更が頻繁に起こるような適応的なシステムを主として対象としており，そもそも機能拡張に対応するための技法であるテスト駆動開発は必須であった．元々 XP の実践項目であったことから，テスト駆動開発はペアプログラミング（Pair Programming）や YAGNI（You Aren't Going to Need It）などの他の XP の実践項目と組み合わせて運用されるのが効果的であるとされている．

なお，テスト駆動開発では外部的な振る舞いに着目してテストを書き，それを実現するように開発することから，振る舞い駆動開発（Behavior-Driven Development（BDD））と呼ばれることもある．

テスト駆動開発のプロセス

テスト駆動開発での開発手順は**図 -1** のようになる．ここでは，与えられた年がうるう年であるかを判定するメソッドを実装するという例を使って説明を試みる．うるう年の判定ルールは以下のようなものである．

ルール 1. 西暦年が 4 で割り切れる年はうるう年
ルール 2. ただし，西暦年が 100 で割り切れる年はうるう年ではない
ルール 3. ただし，西暦年が 400 で割り切れる年はうるう年

① テストケース実装

図 -1 のステップ①ではまず実装する部分を決め，そのためのテストケースを書く．実装する部分をどう選ぶかであるが，1 つのテストケースで押さえられる程度の大きさの機能と考えるとやりやすいかと思われる．1 つのメソッドに複数の機能があるような場合は，複数のテストケースを作成して①～④を繰り返すことになる．

たとえばうるう年判定メソッドには，少なくとも 4 つのテストケースが考えられる．ルール 1 を満足するテストケース，たとえば 1624 に対して戻り値が true．ルール 2 を満足するテストケース，1500 に対する戻り値として false．さらに，ケース 3 を満足するテストケース 1600 に対する戻り値 true．そしてそれ以外の場合のテストケース 1623 に対する戻り値 false である．これらをまとめると**表 -1** のようになる．まず，ここではルー

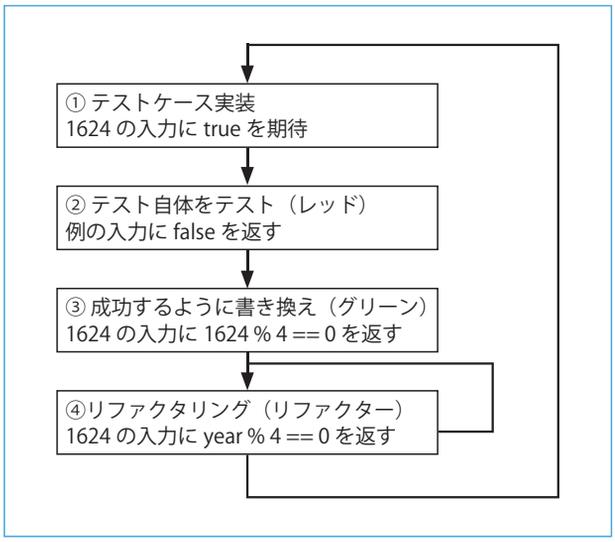

図 -1　テスト駆動開発プロセス

ル 1 用のテストケースに対して実装を行う．

② テスト自体をテスト

ステップ②では，ステップ①のテストケースが通らない実装コードを書き，テスト自体のテストを行う．例のうるう年判定メソッドの場合，「return false;」のように確実に失敗する実装を書いてテストを走らせ，失敗するのを確認する．このように失敗を確認するのは，いきなりテストを成功させてしまうと，たとえ失敗するような実装でも失敗しないというコードになっているかもしれないためである．まずは失敗する実装でテストが正常に動いていることを確認するわけである．

③ 成功するように書き換え

ステップ②でテストが正常に動いていることを確認後，ステップ③としてテストケースを通すように実装コードを書き換える．最初の実装コードは入力値をそのまま流用して書いてみる．入力が 1624 ならばこれを用いてルール 1 のロジックを記述してみる，つまり「1624 は 4 で割り切れる」＝「return 1624 % 4 == 0;」と記述する．ここからの拡張は工程④のリファクタリングで行う．

④ リファクタリング

ステップ④は実装コードを変換していって，妥当なものにする．ステップ③の実装コード内の 1624 は入力値に置き換えることができるので，たとえば入力変数が year だったならば「return year % 4 == 0;」のように書き換えられる．この書き換えを行ったら，テストを実行し書き換えに問題がないかを確認する．数回書き換えが必要となる場合はそのたびにテストを実行して確認する．

十分リファクタリングが終わったと納得後，再びステップ①に戻り現メソッドへの新しい機能の追加や別メソッドの実装へ移る．このうるう年判定メソッドの場合は表 -1 の残りのテストケースの実装を行うことになる．

場合	入力値	戻り値
ルール 1	1624	true
ルール 2	1500	false
ルール 3	1600	true
それ以外	1623	false

表 -1　うるう年判定メソッドのためのテストケース

このようにテスト実行は頻繁に行われるので，テスト駆動開発においては，JUnit のような支援ツールが必須となる．JUnit の GUI では，②の失敗が赤，③での成功が緑で表示されたことから，②→③→④の過程を「レッド，グリーン，リファクター」と呼び，ワルツのリズムのようにリズミカルな実装が行われることになる．ちなみに，このリズミカルさがプログラミングの「楽しさ」につながると言われている．

テスト駆動開発の支援ツール

テスト駆動開発を支援するツールとして代表的なのは，テストを支援するための xUnit フレームワークと，リファクタリングを支援するためのリファクタリングツールである．以下ではこれらを簡単に紹介する．

［xUnit フレームワーク］

前者の xUnit は上述の K. Beck と E. Gamma が開発した JUnit が元になって開発された，複数の単体テストフレームワーク群のことを言う．現在，JUnit, CUnit, CppUnit, NUnit, VBUnit, RubyUnit, PerlUnit, PHPUnit など，各言語に対応した単体テストフレームワークが作られている．ここに名前を挙げたものは，オープンソースで提供されている．

xUnit では，テスト対象モジュールのインタフェースを駆動するためのドライバ・モジュールを構築するフレームワークを提供する．テスト対象のロジックに対するテストケースさえ記述すれば，あとのテスト実行，ログ出力などの定型作業部分はすべてフレームワークで実行される．

たとえば，前述のうるう年判定メソッドのテストケースについてまず，ルール 1 に対するテストケースを記述した時点のものを**図 -2** に示す．オブジェクト指向ということで判定される西暦年は MyYear オブジェクト aYear に格納され，そのオブジェクトに判定メッセージ isLeap() が送られ真偽値が返されるものとする．

JUnit では，1 つのテストケースにつき 1 つのメソッドを対応させ，1 つのクラスに対するテストケースすべてを 1 つのテストクラスに集約するのが慣例となって

```
import junit.framework.TestCase;

// junit.framework.TestCase を継承する
public class MyYearTest extends TestCase {

    // テストケースに対応するテストメソッド
    public void testRule1() {
        // 西暦年を格納するオブジェクトの生成
        MyYear aYear = new MyYear(1624);
        // 判定した結果が真になるべきとの宣言
        assertEquals(true, aYear.isLeap());
    }
}
```

図 -2　うるう年判定メソッドのテストケース記述例

```
public class MyYear {
    // 初期実装では空
    public MyYear(int year) {
    }

    // テストが失敗するように false を返す
    public boolean isLeap() {
        return false;
    }
}
```

図 -3　うるう年判定メソッドの初期実装

```
> java junit.textui.TestRunner MyYearTest
.F
Time: 0
There was 1 failure:
1)
testRule1(MyYearTest)junit.framework.AssertionFailedError: expected:<true> but was:<false>
        at MyYearTest.testRule1(MyYearTest.java:11)
        at sun.reflect.NativeMethodAccessorImpl.
invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.
invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.
invoke(DelegatingMethodAccessorImpl.java:25)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

図 -4　コマンドラインでのテスト実行結果

いる．つまり，ルール 1 のテストケースは，メソッド testRule1 というテストメソッドに書かれ，そのほかのテストメソッドとともに MyYearTest というクラスにまとめられる．

テストメソッドの名前には，test という接頭語をつける以外の制限はない．さらに，JUnit4 では Java5 の言語機能である Annotation を用いることにより，この test という接頭語をつける制限もなくなった．ただし利便性を考えれば，テストメソッドの名前は，そのメソッドでどのようなテストをしているのかが分かるように書くのが望ましい．ここではルール 1 に対するテストであることが分かるように testRule1 という名前をつけている．

このテストケース実装に対する図 -1 工程②の実装を**図 -3** に示す．工程②ではテストをテストするので，isLeap() メソッドが false を返すようにコードを書く．またこの時点ではコンストラクタ MyYear（int）の実装も空でよい．

このコードをコマンドラインおよび付属の GUI で実行した結果は**図 -4** および**図 -5** のようになる（JUnit3.8.1）．

総テスト件数，うちエラー数と失敗数，実行時間，失敗やエラーがあった場合その失敗およびエラー内容，どこでそれらが起こったかが報告される．コマンドラインでは実行，失敗，エラーはそれぞれ「.」，「E」，「F」で表示される．一方，GUI では全部成功であればバーが緑，1 つでも失敗があればバーが赤で表示される．

今回は，1 つのテストメソッドの実行を行い，その 1 つの結果が失敗で，エラーはなかった．そしてその失

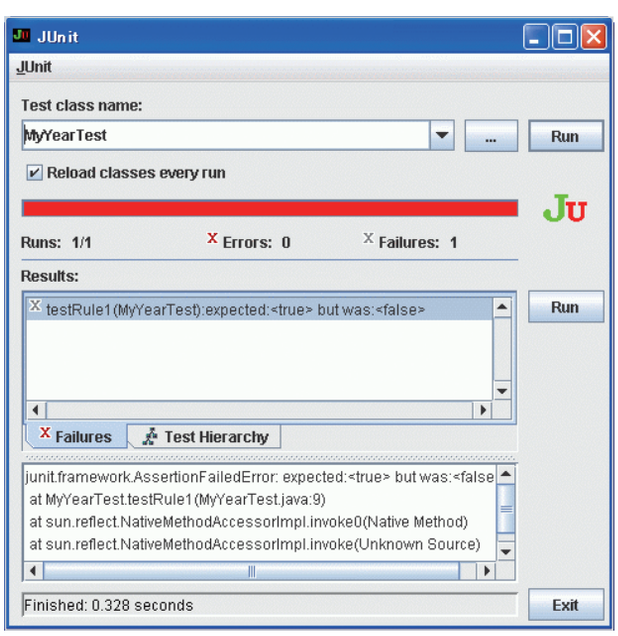

図 -5　GUI でのテスト実行結果

6 Test-Driven Development（テスト駆動開発）開発手法としてのテスト

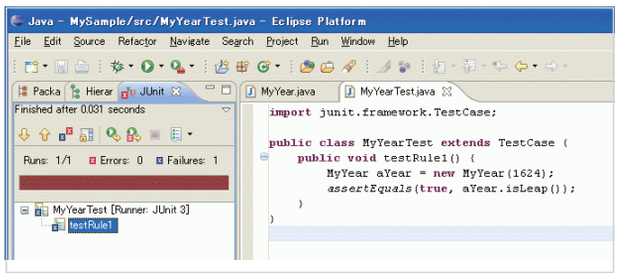

図 -6　Eclipse から呼び出される JUnit 3.8

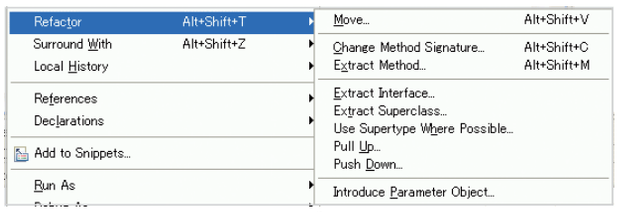

図 -7　Eclipse のリファクタリングメニュー

敗が MyYearTest.java の 11 行目で起こっていて，戻り値が true であることが期待されていたのに false であったと報告されている．

このようにテストケース実行とその後の報告処理はすべて JUnit フレームワークが行ってくれるため，プログラマは基本的にテストケースに対応するテストメソッドの部分だけを書けばよくなっている．

さらに，Eclipse や NetBeans，jEdit のような IDE には JUnit を組み込むようなプラグインが用意されており，エディタやエクスプローラと連携することで，テスト，実装，デバッグが連続的に行える．

図 -6 は Eclipse でテストを実行した画面になる．左側にテストケースの実行数，そのうち何件が失敗し，何件がエラーになったかが表示され，1 件でも失敗かエラーがある場合はバーが赤く，全部成功した場合はバーが緑になる．

その下にはクラスのエクスプローラがあり，テストクラスの中のテストメソッドへ直接飛んだり，またテストメソッド内の呼び出し行から直接呼び出されているテスト対象メソッドへ飛んだりすることができる．たとえば，上述の例での MyYearTest クラスの testRule1() メソッドを左のクラスエクスプローラでクリックして右のエディタにその部分を表示させたり，そこから，isLeap() メソッドをクリックして MyYear クラスの当該コードを表示させたりすることができる．

また，テストクラスの定型コード生成機能も用意されているので，さらに作業が軽減される．具体的には import 文やテストクラスの宣言部，初期化や終了処理のためのメソッドが自動生成される．

ちなみに，JUnit フレームワークはそれ自身がテスト駆動開発で開発されており，その設計はデザインパターンが効果的に使われて構成されている．規模が比較的小さいため，デザインパターンを使った設計を学びたい方には，一度眺めてみることをお薦めする．

さらに，データベースを扱うクラスのテストをするための DBTest のように xUnit を拡張したツールや，擬似的な動作をしてスタブ・モジュールとして機能するようなさまざまな Mock クラス生成ツールのような外部ツールも開発されている．これらの支援ツールを組み合わせて活用することで，効率よくテスト駆動開発を行うことができる．

［リファクタリングツール］

リファクタリングツールは，元々 Smalltalk の開発・実行環境内に用意されていたリファクタリングブラウザと呼ばれる仕組みが一般化したものである．その経緯からリファクタリングブラウザと呼ばれることもある．

リファクタリングツールは，変数名やクラス名などの名称の変更，コードの一部のメソッドとしての抽出，メソッドのクラス間移動，クラス階層への新クラスの挿入などを支援する．実現している機能に幾分のばらつきがあるが，Eclipse などの一部 IDE，Ruby Refuctoring Browser, Xrefactory などがオープンソースで提供されている．

図 -7 は Eclipse が提供するリファクタリング機能のメニューである．改名，移動，メソッドシグネチャの変更，インタフェースやサブクラスの抽出などがサポートされていることが分かる．

大幅な変更を行う場合はテスト駆動開発で用意したテストケース群はチェックに非常に役に立つ．このように，リファクタリングとテスト・ファーストは互いに補い合う関係にある．

テスト駆動開発と従来テスト

テスト駆動開発は，冒頭で述べたように，ソフトウェアテスト技法の 1 つではない．xUnit を単体テストツールとしてテスト工程で使うことは可能ではあるが，xUnit が本来テスト駆動開発で設計支援のために使用されることを意図して設計されているため，テストにおいては必ずしも効率的ではない．たとえば，単体テストであるテストケースを何種類か入れ替えて試したいという場合には，いちいち対応するテストメソッドを探し出しコードを書き換えなければならない．このような用途に対応する場合は外部ツールを導入する必要がある．

テストケースを考えるためには，対象となる機能をよ

く考える必要がある．どのような事前条件で呼び出され，実行が終わったときにはどのような事後条件が成り立っていなくてはならないのか？　より具体的には，どのような引数をとり，有効な値は何で，無効な値には何があり得るのか？　無効な値が与えられたときに，どのような振る舞いをするべきなのか？　具体的な値を与えながら，そのような振る舞いを考えることで，実装対象の機能が，非常に明確になってくる．

このように明確に実装対象の機能を把握することができたなら，実装は非常に楽になる．そして，前もっていろいろなケースを想定しておくことで，思いがけない落とし穴にはまる確率を下げることができる．多くのバグは，実装すべき機能を理解していないか，誤解しているか，そもそも考えもしなかったところに発生すると考えられるからである．

このように，テスト駆動開発におけるテストは設計の手段として用いられるもので，ソフトウェアテストが「バグを見つけること」を目的としているのとは大きく異なる．もちろん，効率的なテストケースの選び方などは，ソフトウェアテストに学ぶところが多くある．同値分割や境界値分析は基本的な技法として知っておく必要があるだろう．また，ソフトウェアテストの領域でも，従来設計の段階からテストを考えていくべきである，という考え方がある．開発工程ごとに住み分けるのではなくプログラマとテスターが互いに知識共有を図って品質を上げる協力をしていく必要があるだろう．

テスト駆動開発の効果

テスト駆動開発はソフトウェア開発にどのような効果を及ぼすのか，これまでさまざまな議論が行われてきた．その一部はプログラマに対する精神的な効果についての議論である．その効果は定性的であり，定量的な評価が難しい．また，その効果は個人の能力や教育などに大きく影響される．一方，その精神的な効果がソフトウェアの品質や生産性に本当に繋がっているのか，という調査も行われてきた．ここではそれらについて簡単に紹介を試みる．

［プログラマに対する精神的な効果］

テスト駆動開発のプログラマに対する精神的な効果としては，次のようなものが挙げられている．

プログラミングが楽しくなる

大きな課題を塊で抱え込むのではなく，分割してテストケースとしてひとつひとつ着実に解決していけることがこまめな達成感を提供する．また，その作業がxUnitの支援下でリズミカルにできるので，プログラミングが楽しくなると言われている．

変更する勇気が出る

すでに動いているコードを何の確認手段もなく変更するのは怖い．しかし，十分に用意された単体テストで個々のモジュールがきっちり押さえられていれば，変更してもどこに影響が出たのかすぐに分かる．これにより，コードを変更する勇気が出ると言われている．

テストしやすいコードを書くようになる

テストを先に書くことから，おのずからテストしにくい実装を避けるようになる．つまり，自然に単機能で他のモジュールとの結合度の低いコードを書くようになると言われている．

［品質と生産性についての調査］

2003年のB. GeorgeとL. Williamsの研究[5]以来，テスト駆動開発によってソフトウェアの品質および生産性が向上するかどうかについて，いくつもの調査研究が行われてきた．結論から言えば，その結果は向上するとするもの，向上しないとするもの，どちらとも言えないとするもの，と割れており，どれが正しいとも言えない状況である．

文献5）では，ウォーターフォール型の開発方式と比較して，ペアプログラミングを用いたテスト駆動開発では18%の品質向上（ブラックボックステストの通過率上昇），16%の開発時間の増加が見られた，と報告された．一般的にテスト駆動開発は，品質向上には有効であるが，人的コストは増加するという傾向が見られるようである．

テスト駆動開発の適用範囲

テスト駆動開発は，アジャイル開発プロセスの一部として，ビジネスアプリケーションの開発現場，特にWebアプリケーション開発現場から生まれてきた．このため，規模が小さく，開始時に要求が明確ではないようなプロジェクトで有効であると評価されてきた．

単体テストにより実装工程で発見的に設計をしていくことから，まずモデルの設計を行った後実装を行う設計駆動型といわれる開発プロセスでは本質的に適用が難しい．実装前にあらかじめ設計を行っていても，テスト駆動開発でリファクタリングを行っていくと，設計文書と乖離してしまうからである．テスト駆動開発では作成されたテストケース群が設計文書的な役割を果たすことになる．

ただし，モデル駆動開発のように抽象レベルを上げて，モデル設計段階にテスト駆動で発見的にモデル構築をす

るという考え方は，将来的にあるかもしれない．

テスト駆動開発では，単体テストでの自動テストを行うことから，GUI のように自動テストしにくい部分や，リアルタイム性が要求される組込み系への適用は難しいとされていた．しかし，近年は GUI の自動テストツールや，組込みシステムのシミュレータの充実などにより，こうした分野での適用例も増えている．

また，データベーススキーマの設計や，BNF で記述された言語の設計など，手続き型言語外のものにも向かないと言われていた．しかし近年，少なくともデータベースに関しては，Test-Driven Database Development（TDDD）といった取り組みもされてきている．

GUI，組込み系，データベーススキーマ設計でのテスト駆動開発については，文献 3）の特集記事にそれぞれ記事があるのでご参照願いたい．

テスト駆動開発のススメ

本稿では，テスト駆動開発がどのようなものであるかを説明し，その効果や適用範囲について現状の紹介を行ってみた．非常に駆け足かつ舌足らずではあったが，テスト駆動開発がどのようなものかが多少なりとも伝わっていれば幸甚である．

テスト駆動開発は日本では早期に紹介されたにもかかわらず，開発現場ではあまり適用されていないようである．先に述べたように，従来の設計駆動型開発プロセスにテスト駆動型開発をそのまま組み込むと設計文書と実装との乖離が生じてしまう．このため，テスト駆動開発の導入にはアジャイル開発プロセスの導入が望ましいが，まだまだ日本の開発現場に受け入れられていない．設計文書による見積もりを出しにくいアジャイル開発プロセスをどうやって導入していいのか分からないという現場の意見を聞いたことがある．

しかし現実的に，製品の仕様変更が頻発し，納期遅れや品質低下が問題になっている．職業人として憂慮すべき状況であり，なんらかの改善を図らなければならない．テスト駆動開発はその対策として十分有益であると考える．テスト駆動開発で変更に対する準備をしておくことで，たとえ仕様変更が起こったとしても，納期遅れや品質低下を避けることができる．

テスト駆動開発は本来的にはアジャイル開発プロセスの思想の上で運用されるべきである．見積もりを出しにくいという問題には，機能単位で見積もるというやり方が提案されている．しかし政治的な理由で，どうしてもアジャイル開発プロセスの導入が難しい場合はテスト駆動開発だけでも取り入れることもやむを得ないとは思う．この場合は設計文書との整合性維持に注意すべきであろう．たとえば RUP（Rational Unified Process）においてテスト駆動開発を取り入れている例はある．総務省が提供している PBL（Project Based Learning）教材[6] にそのような例があったので，参考にされるといいだろう．

参考文献

1）ケント・ベック，長瀬嘉秀（監訳）：テスト駆動開発入門，（株）ピアソン・エデュケーション（2003）．
2）JUnit.org, http://www.junit.org/
3）TDD : The Art of Fearless Programming, IEEE Software, Vol.24, No.3, pp.24-83 (2007).
4）マーチン・ファウラー，児玉公信他（訳）：リファクタリング，（株）ピアソン・エデュケーション（2000）．
5）George, B. and Williams, L. : An Initial Investigation of Test-Driven Development in Industry, ACM Symposium on Applied Computing (SAC), Melbourne, FL, pp. 1135-1139 (Mar. 2002).
6）高度情報通信人材を育成する実践的なプログラム教材の開発，総務省報道資料，http://www.soumu.go.jp/s-news/2007/pdf/070522_2.pdf

（平成 19 年 12 月 28 日受付）

大月美佳（正会員）

mika@is.saga-u.ac.jp

平成 11 年，九州大学大学院システム情報科学研究科知能システム学専攻博士後期課程修了．博士（工学）．佐賀大学理工学部知能情報システム学科講師．主研究テーマは，ソフトウェア開発技術およびネットワークを利用した教育技術．NPO ASTER 理事．日本図学会，ゲーム学会各会員．